AL POSTGRADUATE SCHOOL
TEREY, CALIFORNIA 93943-5002

NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

A MULTIMEDIA COMPUTER CONFERENCING SYSTEM

by

James Edward Manley

December 1986

Thesis Advisor: Michael Zyda

Approved for public release; distribution is unlimited.

T230816

unclassified
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a REPORT SECURITY CLASSIFICATION unclassified	1b RESTRICTIVE MARKINGS
2a SECURITY CLASSIFICATION AUTHORITY	3 DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution is unlimited.
2b DECLASSIFICATION/DOWNGRADING SCHEDULE	
4 PERFORMING ORGANIZATION REPORT NUMBER(S)	5 MONITORING ORGANIZATION REPORT NUMBER(S)
6a NAME OF PERFORMING ORGANIZATION Naval Postgraduate School 6b OFFICE SYMBOL (If applicable) 52	7a NAME OF MONITORING ORGANIZATION Naval Postgraduate School
6c ADDRESS (City, State, and ZIP Code) Monterey, California 93943-5000	7b ADDRESS (City, State, and ZIP Code) Monterey, California 93943-5000
8a NAME OF FUNDING/SPONSORING ORGANIZATION 8b OFFICE SYMBOL (If applicable)	9 PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c ADDRESS (City, State, and ZIP Code)	10 SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO / PROJECT NO / TASK NO / WORK UNIT ACCESSION NO

11 TITLE (Include Security Classification)
A MULTIMEDIA COMPUTER CONFERENCING SYSTEM

12 PERSONAL AUTHOR(S) Manley, James Edward

13a TYPE OF REPORT	13b TIME COVERED	14 DATE OF REPORT (Year, Month, Day)	15 PAGE COUNT
Master's Thesis	FROM _____ TO _____	1986 December	87

16 SUPPLEMENTARY NOTATION

17 COSATI CODES			18 SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	multimedia computer conferencing; multicomputer network information-sharing; command and control systems; integrated office automation systems

19 ABSTRACT (Continue on reverse if necessary and identify by block number)

The purpose of this study is to review the state of interworkstation computer communications, suggest ways in which these communications can be utilized for multimedia computer conferencing, and provide the details of a prototype system which demonstrates some of the capabilities that multimedia computer conferencing systems can have. The source code for the prototype system is provided in the appendices.

The results of this study are of interest to people designing command and control systems, integrated office automation systems, and personal and business communications systems. Additionally, the details of the multicomputer network information-sharing portion of the prototype can be useful to people who desire to distribute computational loads over a number o systems such as distributed graphics processing, scientific modeling and simulation, and engineering design and prototyping.

20 DISTRIBUTION/AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT ☐ DTIC USERS	21 ABSTRACT SECURITY CLASSIFICATION unclassified
22a NAME OF RESPONSIBLE INDIVIDUAL Prof. Michael J. Zyda	22b TELEPHONE (Include Area Code) (408) 646-2305 22c OFFICE SYMBOL Code 52Zk

DD FORM 1473, 84 MAR 83 APR edition may be used until exhausted
All other editions are obsolete

SECURITY CLASSIFICATION OF THIS PAGE
unclassified

Approved for public release; distribution is unlimited.

A Multimedia Computer Conferencing System

by

James Edward Manley
Lieutenant Commander, United States Navy
B.S., United States Naval Academy, 1976

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
December 1986

ABSTRACT

The purpose of this study is to review the state of interworkstation computer communications, suggest ways in which these communications can be utilized for multimedia computer conferencing, and provide the details of a prototype system which demonstrates some of the capabilities that multimedia computer conferencing systems can have. The source code for the prototype system is provided in the appendices.

The results of this study are of interest to people designing command and control systems, integrated office automation systems, and personal and business communications systems. Additionally, the details of the multicomputer network information-sharing portion of the prototype can be useful to people who desire to distribute computational loads over a number of systems such as distributed graphics processing, scientific modeling and simulation, and engineering design and prototyping.

TABLE OF CONTENTS

I. INTRODUCTION 9

A. EXISTING COMMUNICATIONS METHODS 9

B. MOTIVATION FOR MULTIMEDIA COMPUTER CONFERENCING 10

C. MULTIMEDIA COMPUTER CONFERENCING NOW FEASIBLE 10

1. Graphics Hardware 10

2. Audio Hardware 11

D. MULTIMEDIA COMPUTER CONFERENCING CAPABILITIES 12

E. OVERVIEW OF THE REMAINING CHAPTERS 13

II. BACKGROUND 14

A. RELATED COMPUTER DEVELOPMENTS 14

1. Hardware 14

2. Software 16

B. PREVIOUS WORK ON MULTIMEDIA COMPUTER CONFERENCING 17

C. STANDARDS 18

III. SPECIFICATIONS 20

A. FUNCTIONAL REQUIREMENTS 20

1. Network File Transfer Mechanism 20

2. Multitasking 21

3. Hierarchical Filing Structure 21

B. DESIRABLE CHARACTERISTICS 22

1. High-Resolution, Three-Dimensional Graphics 22
2. Flexible User Interface 22
3. Multiple Windows 23
4. Graphical Data Flow 23
C. HYPOTHETICAL MULTIMEDIA COMPUTER CONFERENCING SESSION 24
1. Activation of a Multimedia Computer Conferencing Session 24
2. Interaction Among Conference Participants 25
3. Intensive Two-Way Communications 25
IV. IMPLEMENTATION – COMMUNICATIONS PACKAGES 26
A. HARDWARE 26
B. COMMUNICATIONS NETWORK 27
1. Topology 27
2. Performance 27
C. SOFTWARE 27
1. Operating System 27
2. Communications Packages 28
a. Sockets 29
b. Client/Server Model 30
c. Using the Communications Packages in Applications ... 31
d. Details of the Communications Packages 33
(1) UNIX 4.2 BSD Communications Package 34
(a) Opening a Server Socket 34
(b) Opening a Client Socket 38
(c) Reading and Writing Data 39

(2) UNIX System V Communications Package 41
V. APPLICATIONS PROGRAMS 43
A. INTERTEXT 43
B. INTERDRAW 44
VI. LIMITATIONS AND CONCLUSIONS 47
A. LIMITATIONS 47
1. Synchronous Communications 47
2. Only Text and Graphics Media Currently Supported 48
B. CONCLUSIONS 48
1. Problem Areas 48
a. UNIX 48
b. Network Communications 49
c. Multicomputer Programming Tools 49
2. Recommendations 50
a. Standard Data Formats 50
b. Advanced User Interfaces 51
c. Enhancements to the Prototype Multimedia Computer Conferencing System 51
(1) Object Manipulation Applications 51
(2) Speech Recognition and Synthesis 52
APPENDIX A UNIX 4.2 BSD COMMUNICATIONS PACKAGE LISTING 53
APPENDIX B UNIX SYSTEM V COMMUNICATIONS PACKAGE LISTING 61
APPENDIX C INTERTEXT PROGAM LISTING 69
APPENDIX D INTERDRAW PROGRAM LISTING 72
LIST OF REFERENCES 84

BIBLIOGRAPHY 85
INITIAL DISTRIBUTION LIST 86

ACKNOWLEDGEMENT

I would like to thank my thesis advisor, Professor Michael J. Zyda, for his inspiration, guidance, and patience throughout the duration of this project. Without his assistance, this study would still be in progress.

I also want to recognize the programming assistance provided by Mr. Albert Wong of the Naval Postgraduate School Computer Science Department. His expertise in UNIX systems programming proved invaluable in making the prototype system a reality.

To the other Naval Postgraduate School faculty members whom I have studied under, I extend my warmest regards and gratitude for the help they have given me in better understanding the true meaning of scientific exploration.

Finally, I want to express my deepest appreciation and affection to Emma, my wife, for her understanding and the sacrifices she made to allow me to pursue this work. Her contributions were unquestionably the most important of all.

I. INTRODUCTION

A. EXISTING COMMUNICATIONS METHODS

Communication is an important capability for humans since it is the means by which we convey information and ideas, thereby expanding our knowledge, culture, and technology. Several means of communication are used by people in their daily activities including voice, writing, drawing, gestures, facial expressions, and body movements. These methods can be used either in person or over any distance via electronic media. Voice and writing can be disseminated by telephone and radio systems, and television can transmit all six communication methods. However, most people do not normally have access to television for personal communications. This is even more true in situations where information is to be originated by and shared between more than two people. Therefore, there are currently only three methods in widespread use that enable people to interactively share information: face–to–face, by telephone, and by teletype/computer terminals. However, the distance at which the first method can be used is limited to a few tens of meters, and the speed at which information can be transferred between people via the latter two methods is limited because of their inherently low bandwidths. Visually oriented communications contain a much higher bandwidth of information, but require correspondingly faster transmission channels such as television broadcast frequencies for acceptable real–time communications. Various methods to encode visual information for transmission over low–bandwidth media

have been developed, but the side–effects of the encoding processes and limitations in terminal hardware have resulted in low quality visual displays.

B. MOTIVATION FOR MULTIMEDIA COMPUTER CONFERENCING

During the past four decades, computers have become powerful tools used in one form or another by a large percentage of the world's population. The capabilities of these tools have increased incredibly and their accessibility has grown dramatically. Despite advances in individual computer power and the exponential growth in the number of computers in operation, the ability for people to easily communicate abstractions via different computers has not matured as rapidly. Many systems still use a line–oriented textual input/output mechanism used on the earliest computers even though the display medium has evolved from punched tape and cards to cathode ray tubes.

C. MULTIMEDIA COMPUTER CONFERENCING NOW FEASIBLE

1. Graphics Hardware

The development of low–cost, high–density processor and memory devices enabled the mass production of high–resolution bit–mapped raster graphics displays. Bit–mapped raster graphics allows both images and text to be displayed concurrently in any mixture, location, size, style, orientation and intensity or color on a display device. This visually–oriented capability allows the implementation of the window/icon/mouse concept, where programs, data, and devices can be represented by images in addition to text.

High–resolution bit–mapped graphics and advances in microprocessor technology provided the raw materials needed for the creation of graphics workstations and advanced microcomputers with performance equal to that of minicomputers. The integration of special–purpose coprocessors for floating point arithmetic, matrix manipulation, and video display into these systems has resulted in graphics processing performance unequaled by any other type of computer.

The advent of high–performance graphics workstations has made it possible to reduce the amount of communications bandwidth required to transfer high–resolution graphics information. Instead of transferring entire bit–mapped images frame–by–frame over networks, the graphics processing power of these machines allows just the characteristics of a graphics object to be transmitted while the receiving system can reconstruct the object very quickly, usually in less than one video frame interval (1/30th of a second). Once an object's characteristics have been received by a remote workstation, it is then possible to send commands to that workstation to translate, scale, and rotate objects. With this capability, it is possible for people to interactively manipulate graphics entities in real time.

2. Audio Hardware

Along with visually–oriented information formats, the ability to transmit audio information can be supported. Some microcomputers now possess the capability to digitize and reproduce sound very accurately as well as convert text to speech. With this added feature, it is possible to

implement a multimedia computer conferencing system which allows two or more people to quickly share audio as well as visual information.

D. MULTIMEDIA COMPUTER CONFERENCING CAPABILITIES

A multimedia computer conferencing system provides the ability for two or more people to share information in textual, graphic and audible form simultaneously. Textual and graphics information is displayed in windows on a workstation screen while audio information is produced through a microcomputer. Users can select objects on their display screens using a pointing device such as a mouse and cursor icon and perform operations on them such as translation, scaling, rotation and modification. These operations can be duplicated on other users' display devices in real–time and a number of users can manipulate objects in a window interactively to illustrate ideas. While graphics operations are carried out on objects, text typed by users can be converted to speech so that other users can watch the graphics while listening to a user's ideas. Digitized sound can also be transmitted as desired to enhance the realism or impact of graphics presentations. All of these actions can be recorded as they are produced or received so that all or any portion can be played back for people unable to participate in a live session or to study details more closely. These recordings can also be edited to improve a presentation, particularly if it is to be given at a later time to a large number of people in one or many locations.

```
remote_server_socket =
    connect(local_socket, (caddr_t)&address, sizeof(address));
```

Although the remote server system socket number is returned, the client program must refer to this connection for reading and writing data by its local socket number. This is in direct opposition to the server, which refers to the connection by the remote socket number it received from accept().

(c) Reading and Writing Data. At this point, the accept() function in the server allows execution to resume, and a communications connection has been established between the server and client programs. This connection is two–way in nature, so that data can be transferred in either direction. However, the UNIX functions which send and receive data, the write() and read() functions, respectively, also block execution of a program in which they are called until each data transfer is complete. A method that can be used to circumvent this problem in the program sending data is to start a child process with the fork() function and make the call to the write function within the child process. This allows the program sending information to continue execution even if the write() function call is blocked. The write_immediate() function in the communications package performs this operation.

```
write_immediate(socket_number, buffer_pointer, nbytes)

    /* socket number to be written to */
    int socket_number;

    /* buffer of bytes to be written */
    char buffer[];

    /* the number of bytes to be written */
    long nbytes;
```

```
{

/* function which initiates a child process */
int fork();

/* value returned from routine fork()  */
int forkval;
```

First, initiate a child process which will perform the write operation.

```
forkval = fork();
```

Next, determine if a child process was started and if so, begin writing data.

```
/* check to ensure a zero was returned by fork() indicating a child
   process was started */
if(forkval == 0)
   {
   while(write(local_socket,buffer,nbytes) != nbytes)
      {
      /* perform the write operation */
      }
```

When the data has been written, terminate the child process.

```
   exit(1);
   }
```

Otherwise, return an error value to the application indicating that the child process was not successfully started and therefore the data was not written.

```
else
   {
   return(-1);
   }
return(forkval);
} /* end of function write_immediate() */
```

If the write was successful, the value of the forked process is returned to the application program to indicate the successful write operation.

An attempt was made to create an asynchronous read_immediate() function, but the absence of an interrupt–driven interprocess communications capability in UNIX prevented the achievement of such a function. The designers and implementers of UNIX 4.2 BSD at the University of California at Berkeley have considered providing such a feature, but have not yet done so. In the meantime,

further research into the implementation of an asynchronous read function is needed, possibly utilizing semaphores or another low–level operating system mechanism.

(2) UNIX System V Communications Package. The communications package for UNIX System V is very similar to that for 4.2 BSD, so only the differences will be explained. The acquisition of a local socket for clients and servers under System V is the same as that for 4.2 BSD with a few exceptions. First, the System V rhost() function is called instead of the 4.2 BSD gethostbyname() function to convert a system name to its network address. The same double–indirected pointer to the name is used as an argument.

```
remote_client_address = rhost(&ptr_client_name);
```

Calling the socket() function in System V is different than calling socket() in 4.2 BSD. The type of data packet delivery (SOCK_STREAM or SOCK_DGRAM) is passed as the first parameter. Next, the protocol type structure is passed, which must be type–cast to the sockproto structure if the default value of zero is used. Note that a significant difference in the System V socket() function call is that a pointer to the Internet socket address data structure is passed to the socket() function directly as the third parameter. This is the reason that there is no bind() function in System V like that in 4.2 BSD. Finally, an options parameter that is unique to System V is provided which specifies how the socket is to be maintained in case an attempt to accept a connection fails.

```
int local_client_socket;

 local_client_socket =
    socket(SOCK_STREAM, (struct sockproto *)0, &address, options);
```

Once a local socket is established, a connection can be accepted by a server or connect() can be called by a client program. In both cases, the local socket number and a pointer to the Internet socket address data structure are passed as parameters and an integer representing the remote socket number is returned. If the value returned is less than zero, an error occurred in the accept() or connect() call. In that case, the local socket should be closed with a call to close(local_socket) and another local socket should be opened before calling either accept() or connect().

```
remote_client_socket = accept(local_socket, &address);
                            or
remote_server_socket = connect(local_socket, &address);
```

If the remote socket number is non–negative, then the connection has been established and in both cases the local socket number is returned to the applications program. This is yet another difference from 4.2 BSD because in that system the remote socket number must be used to refer to the socket after accept() is called, while the local socket number is only used after a call to connect().

V. IMPLEMENTATION – APPLICATIONS PROGRAMS

Two applications programs were written to demonstrate the potential capabilities of a multimedia computer conferencing system. Each program demonstrates how information from a single medium such as text or graphics can be transferred. Source code listings for the programs are located in Appendices C and D.

A. INTERTEXT

Intertext is a very simple program which allows two users to establish a connection between themselves using workstations for the exchange of messages. When the program is started, the user is asked whether a session is being started by himself or another user. If the user is originating the session, a system name and port number are asked for, the program starts the connection process and notifies the user that it is waiting for the remote user to connect. If the user is completing a connection to a session already originated by a remote user, the program requests the remote system name and port number and notifies the user when the connection is complete. The user who originated the session should type in a messge first (messages are sent by pressing the "return" key). The message is printed on the remote user's screen and then a response can be sent from the remote user in a similar manner. A session is terminated by either user sending a message which begins with a lowercase 'q'. If for some reason an attempt to establish a connection fails, the program can be run again using a different port number. Users should

set aside a block of port numbers to be used for Intertext sessions and notify remote users which port number to use by another means such as electronic mail.

The IRIS workstations support up to ten windows on a screen at one time and any number of these can be Intertext program windows, so that conversations can be had with up to ten other users at one time. The ISI workstations have a similar capability. The terminals connected to the VAX systems are text–only display devices and can therefore support only one conversation at a time.

Note in the Intertext listing in Appendix C that only one call to the communications package is made, either to connect_client() or connect_server(), to establish the connection. Simple read() and write() function calls transfer the data and close() is used to terminate the connection.

B. INTERDRAW

Interdraw is a graphics program which allows two users to simultaneously draw straight line segments in one window on each of their graphics workstations. The startup procedure is the same as that for Intertext, with one user being the initiator and the other the responder. After a connection is established, a window is opened on each workstation screen. To draw in the window, the right mouse button is pressed to activate the window. If the mouse is moved, the cursor in the window will follow its movements up, down, left, and right and a "rubber band" line segment will be drawn from the center of the window to the cursor. Pressing the middle mouse button will cause the "rubber band" line

segment to be drawn from the current cursor position. Pressing the right mouse button will draw a permanent line segment from the last place it was pressed to the current cursor location. Each user's line segments are displayed in red on his local workstation while the remote user's line segments are displayed in blue.

As shown in the listing in Appendix D, like Intertext, the only call that Interdraw makes to the communications package are to connect_client() or connect_server(). Once communications are started, the programs alternately call write() and read() functions to send and receive commands and positional data. This information is organized in each transmission in a data structure which holds three floating point or equivalent–sized values. The first value is the action to be taken (no action, draw a new line segment, or move the starting point of the next line segment) and the remaining two values are the x and y coordinates in world space, respectively. In other applications, a more complex data structure can be devised and all that is necessary to send or receive the data is to call the write() function with a socket number, the address of the data structure (a pointer), and the number of bytes in the data structure (using the sizeof(structure–name) function).

Interdraw can be used in parallel with Intertext for a multimedia conferencing session on a multitasking workstation such as the IRIS with each program executing in its own window. This combination demonstrates one of the most important features a multimedia computer conferencing system can offer – interactive information sharing, where people can easily use drawings to convey their thoughts and amplify textual

descriptions. Just as scratchpads, chalkboards and cocktail napkins have been used to make important ideas clear, Interdraw allows people to communicate more effectively.

VI. LIMITATIONS AND CONCLUSIONS

The prototype multimedia computer conferencing system developed in this study demonstrates the feasibility of implementing such a system using existing software and hardware. It is a prototype and therefore does not possess all of the qualities desirable in a full implementation. However, the basic capabilities can be expanded as needed to improve its ability to support transfer of information with existing information processing systems. The following sections discuss some of the systems current shortcomings and recommended avenues for future exploration in the area of multimedia computer conferencing.

A. LIMITATIONS

1. Synchronous Communications

The prototype implementation is somewhat hampered by the synchronous nature of the UNIX interprocess and network communications facilities. Applications must be designed so that users must know whether they are executing the program as a server or a client. If the users at two sites do not properly coordinate the activation of their programs, a connection is not established and the programs must be reinitialized. The synchronization requirement also dictates that both entities on a connection perform a read/write transaction before the program segments they are executed in are allowed to continue execution. This problem can be eliminated by the development of an asynchronous read() function. This is possible to achieve using the shared memory

feature of UNIX System V which allows two or more processes to read and write data in memory on a single processor asynchronously. An asynchronous read() function can most likely be implemented under 4.2 BSD using low–level operating system functions.

2. Only Text and Graphics Media Currently Supported

The Specifications chapter discussed the desirability of transferring audio information during a conference. Although not as significant as text and graphics, this is an important feature and should be provided in a full–scale system.

B. CONCLUSIONS

1. Problem Areas

a. UNIX

Using UNIX as the operating system for multimedia computer conferencing presented some problems. Although the socket mechanism is described in the UNIX documentation as having the ability to connect multiple clients to a given server through one port, no more than one client could be connected to a server at one time through one port on the prototype system. To circumvent this problem, conference server programs have to open a socket for each client which desires to connect to that conference. This consumes more sockets and makes the connection operation much more complicated than would otherwise be necessary. Another problem with UNIX sockets is that the client/server model on which socket connections are based is asymmetric and synchronous, that is, it forces a server program to suspend execution until a client connects to it and processes cannot continue execution until data written by one process is

read by another and vice versa. An interrupt–driven means of exchanging data would be very useful and, although such a capability has been considered by the designers of UNIX, it has not been specified or implemented yet.

b. Network Communications

The central Ethernet network which connects the computer systems together is adequate for file transfers and electronic mail between systems. However, because of the large number of users which work on the systems connected to the network, data transfer rates for programs which are communications–intensive slow down perceptibly when waiting for data. Also, some data was lost, even when the UNIX SOCK_STREAM guaranteed sequential packet delivery option was used. These problems indicate that multimedia computer conferencing should be conducted over communications channels which are not used for large–scale data transfer operations, if possible. Due to the effective data compression possible with multimedia computer conferencing systems, it may be feasible to use dedicated lower–speed communications channels for this purpose.

c. Multicomputer Programming Tools

In any type of programming, powerful tools can make designing, implementing and testing programs much more effective. As in most software environments, the tools available to people working on multimedia computer conferencing systems are inadequate. It is difficult enough to debug a program on a single computer system – it becomes extremely difficult to debug a number of programs executing on two or more computer systems, particularly if they are physically separated by more than a few

feet. Many hours were spent during the development of the prototype attempting to isolate the causes of problems. Frequently, it was not possible to determine which system an error was occurring in and only time–consuming trial–and–error techniques resulted in isolation of some problems. Future work in multimedia computer conferencing will require more advanced programming tools in order to achieve advanced capabilities in these systems. In particular, multisystem debugging aids are needed to help alleviate the problem of isolating errors to a single system and individual program. Information about what data has been transmitted and received and the sequence in which the data was transferred is very important and not readily available with current tools.

2. Recommendations

For systems software engineers who are developing multimedia computer conferencing systems, there are some recommendations which result from the experience of implementing and using this prototype.

a. Standard Data Formats

One of the primary limitations of the prototype developed in this thesis is the rudimentary set of data formats used to identify various types of data such as text and graphics. In a full–scale implementation, a much larger set of formats would be necessary to hierarchically define subtypes of data such as graphics primitives and objects. Some work has already been done by Aguilar [Ref. 3] in developing data formats suitable for use in multimedia computer conferencing. Eventually, a standard set of data formats will have to be agreed upon if multimedia computer conferencing is to become a widely used. The standards process is a long and complex

one which takes years to traverse. One of the keys to early, widespread implementation of multimedia computer conferencing will be provision of a capability to expand the standard since it is extremely difficult to forsee the requirements which will be placed on such systems in future years.

b. Advanced User Interfaces

With the proliferation of hardware and software which provide advanced user interface features, it has become clear that there is more to an effective user interface than windows, menus, and mice. Many systems that are equipped with a graphics interface do not provide a full set of programming tools to allow applications programmers to customize. A number of systems such as the Apple Macintosh, Commodore Amiga, Atari ST, and Digital Research GEM environment have "toolboxes" containing hundreds of routines for user interface functions that can be called from any programming language implemented for the system. These routines make implementing user interface features in programs much easier and more consistent since the programmer doesn't need to build them from scratch. If all systems had a rich set of such routines, then implementing multimedia computer conferencing would be more straightforward and consistent between machines.

c. Enhancements to the Prototype Multimedia Computer Conferencing System

(1) Object Manipulation Applications. The engineering and science communities would benefit greatly from programs which allow people to translate, scale, and rotate objects or components thereof interactively. This is already feasible on a system such as the IRIS, but a standard method of hierarchically defining and referring to objects (such

as the proposed PHIGS standard) is required to allow a large number of people on different workstations to utilize this capability.

(2) Speech Recognition and Synthesis. Although typical workstations do not possess sound digitizing, playback, and speech recognition/synthesis capabilities, many microcomputers do and it is feasible to connect a microcomputer to a workstation to handle the audio medium. The advantage of speech over a text window is that the user can be looking at graphics being transmitted by another user and receiving other information by listening, thereby increasing the overall bandwidth of information transfer over that of only text.

APPENDIX A

UNIX 4.2 BSD COMMUNICATIONS PACKAGE LISTING

This appendix contains source code, written in the C language, which enables communications to be established between processes executing on two UNIX systems (4.2 BSD and/or System V). The source for UNIX 4.2 BSD is provided in this appendix for systems programmers interested in understanding how to utilize the UNIX intersystem communications functions. It is not necessary for applications programmers to know the details of these programs in order to use them to establish communications between programs on different systems. Only the calling conventions for the connect_server, connect_client, and write_immediate() functions need to be understood.

```
/*

Program segment net4.2.c
Version 19 November 1986

This segment, when linked into a program on a computer with a
UNIX 4.2 BSD operating system, will allow the program to
communicate with programs executing on other computer systems
over an Internet network.

*/

#define TRUE 1

/* include files for UNIX 4.2 BSD */

#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>

/*************************************************************
```

```
The connect_server(remote_client_name, port_number) function
performs the actions required to connect a server system to a
remote client system

**********************************************************/


int connect_server(remote_client_name, port_number)

   /* name and port number of the remote client system */
   char remote_client_name[];
   int  port_number;

   {

   /* pointer to the remote client system's name */
   char *ptr_client_name;

   /* local socket number */
   int local_server_socket;

   /* function that opens a socket */
   int socket();

   /* function that accepts a connection from a remote client
      socket */
   int accept();

   /* socket number of the remote client system */
   int remote_client_socket;

   /* protocol and address data structure specified for the
      socket */
   static struct sockaddr_in address  = { AF_INET };

   /* address of the remote client system */
   long remote_client_address;

   /* port number of the remote client system */
   short remote_client_port;

   /* size of the address data structure of the remote client
      system */
   int address_size;


   /* begin the process of attempting to connect to the
      remote client system */

   /* get a pointer to the remote client system's name */
   ptr_client_name = &remote_client_name[0];
```

```
/* convert the remote client system name to its address */
remote_client_address =
   (long)gethostbyname(&ptr_client_name);

/* initialize the remote client port number above the
   system reserved ports */
remote_client_port = IPPORT_RESERVED + 1;

/* add the remote client port number to the number of
   reserved ports */
remote_client_port = remote_client_port + port_number;

/* initialize the remote client socket number to an
   invalid value */
remote_client_socket = -1;

/* remote client system address family (Internet in this
   case) */
address.sin_family = AF_INET ;

/* place the remote client port number into the address
   data structure */
address.sin_port = remote_client_port;

/* put the port number in network byte order */
address.sin_port = htons(address.sin_port);

/* place the remote client system's address in the address
   data structure */
address.sin_addr.s_addr = remote_client_address;

/* find number of bytes in the remote client address */
address_size = sizeof(remote_client_address);

/* attempt to open a local socket */
local_server_socket = socket(AF_INET,SOCK_STREAM,0);

if(local_server_socket < 0)
   {
   /* the server couldn't open a local socket */
   perror("Server couldn't open a local socket:");
   }

else
   {

   if(bind(local_server_socket, (caddr_t)&address,
           sizeof(address)) < 0)
      {
      perror("Server couldn't bind address to local
```

```
                socket: ");
        }

    /* set the maximum number of remote client systems to
       be connected to */
    listen(local_server_socket,5);

    printf("Server waiting to connect to
            %s\n",remote_client_name);

    /* attempt to accept a connection */
    remote_client_socket =
       accept(local_server_socket, &address,
              &address_size);

    if(remote_client_socket < 0)
       {
       /* an error occurred in the server attempting to
          accept a connection from remote client system */
       perror("Server couldn't accept a connection from the
               remote client system: ");

       close(local_server_socket);
       }
    else
       {
       /* the server accepted a connection from the remote
          client system */
       break;
       }

    } /* end else local_server_socket >= 0 */


 /* return the socket number of the remote client system */
 return(remote_client_socket);

 } /* end of function connect_server() */



/************************************************************

The connect_client(remote_server_name port_number) function
performs all the actions required to connect a client system
to a remote server system

************************************************************/

int connect_client(remote_server_name, port_number)
```

```
/* name and port number of the remote server system */
char remote_server_name[];
int  port_number;

{

/* pointer to the name of the remote server system */
char *ptr_server_name;

/* local socket number */
int local_client_socket;

/* function that opens a socket */
int socket();

/* function that connects local socket to remote server
   socket */
int connect();

/* the socket number on the remote server system  */
int remote_server_socket;

/* the protocol and address data structure specified for
   the socket */
static struct sockaddr_in address = { AF_INET };

/* address of the remote server system */
struct hostent *remote_server_address;

/* port number of remote system */
short remote_server_port;


/* begin the process of attempting to connect to the
   remote server system */

/* establish ptr to the remote server name */
ptr_server_name = &remote_server_name[0];

/* convert the name of the remote server system to an
   address */
remote_server_address = gethostbyname(ptr_server_name);

/* clear out the address structure */
bzero((char *)&address, sizeof(address));

/* copy the remote server address structure into the
   address structure */
bcopy(remote_server_address->h_addr,
      (char *)&address.sin_addr,
```

```
      remote_server_address->h_length);

/* initialize remote server port number above the system
   reserved ports */
remote_server_port = IPPORT_RESERVED + 1;

/* add the user's remote server port number to the number
   of reserved ports */
remote_server_port = remote_server_port + port_number;

/*  remote server system address family(Internet in this
    case) */
address.sin_family = AF_INET;

/* place the remote server port number into the address
   structure */
address.sin_port = remote_server_port;

/* put the port number in network byte order */
address.sin_port = htons(address.sin_port);

/* attempt to obtain a local socket */
local_client_socket = socket(AF_INET, SOCK_STREAM, 0);

if(local_client_socket < 0)
   {
   /* the client couldn't open a local socket */
   perror("Client couldn't open a local socket: ");
   }
else
   {
   /* place Internet address family type in address
      structure */
   address.sin_family = AF_INET;

   /* attempt to connect local client socket to remote
      server socket */
   remote_server_socket =
      connect(local_client_socket, (caddr_t)&address,
              sizeof(address));

   if(remote_server_socket < 0)
      {
      /* an error occurred in attempting to connect to
         the remote server socket */
      perror("Client couldn't connect to the remote
              server socket: ");
      close(local_client_socket);
      }
   else
      {
```

```
            /* successfully connected to the remote server
               system */
            printf("Connection established with
                    %s.\n",remote_server_name);
            break;
            }

         } /* end else socket >= 0 */

      /* return the socket number of the local client system */
      return(local_client_socket);

      } /* end of funcition connect_client() */


/*************************************************************

function write_immediate(socket_number,buffer,nbytes) writes
to the network connection by forking a child process which
actually performs the write operation.

*************************************************************/


int write_immediate(socket_number, buffer_pointer, nbytes)

      /* socket number to be written */
      int socket_number;

      /* buffer of bytes to be written */
      char buffer[];

      /* the number of bytes to be written */
      long nbytes;

      {

      /* function which initiates a child process */
      int fork();

      /* value returned from routine fork()  */
      int forkval;


      /* initiate a child process which will perform the write
         operation */
      forkval = fork();

      /* determine whether a child process was successfully
         started */
      if(forkval == 0)
```

```
   {
   /* attempt to perform the write operation */
   while(write(local_socket,buffer,nbytes) != nbytes)
      {
      /* attempt to write the buffer contents */
      }
   /* terminate the child process after the buffer is
      successfully written */
   exit(1);
   }
else
   /* an error occurred in starting the child process */
   {
   perror("Error occurred while attempting to fork a
           write-immediate process: ");
   /* return an error value to the application */
   return(-1);
   }
/* return a value indicating successful operation */
return(forkval);
} /* end of write_immediate() function */
```

APPENDIX B

UNIX SYSTEM V COMMUNICATIONS PACKAGE LISTING

This appendix contains source code, written in the C language, which enables communications to be established between processes executing on two UNIX systems (4.2 BSD and/or System V). The source for UNIX System V is provided in this appendix for systems programmers interested in understanding how to utilize the UNIX intersystem communications functions. It is not necessary for applications programmers to know the details of these programs in order to use them to establish communications between programs on different systems. Only the calling conventions for the connect_server, connect_client, and write_immediate() functions need to be understood.

```
/*

Program segment netV.c
Version 03 November 1986

Link with IRIS system library file /usr/lib/libsocket.a

This segment, when linked into a program on a computer with a
UNIX System V operating system, will allow the program to
communicate with programs executing on other computer systems
over an Internet network.

*/

#define TRUE 1

/* include files for IRIS UNIX System V */

#include "/usr/include/EXOS/sys/socket.h"
#include "/usr/include/EXOS/netinet/in.h"


/*************************************************************
```

```
The connect_server(remote_client_name, port_number) function
performs the actions required to connect a server system to a
remote client system

**********************************************************/


int connect_server(remote_client_name, port_number)

   /* name and port number of the remote client system */
   char remote_client_name[];
   int  port_number;

   {

   /* pointer to the remote client system's name */
   char *ptr_client_name;

   /* local socket number */
   int local_server_socket;

   /* function that returns the address of the remote client
      system */
   long rhost();

   /* function that opens a socket */
   int socket();

   /* function that accepts a connection from a remote client
      socket */
   int accept();

   /* socket number of the remote client system */
   int remote_client_socket;

   /* protocol and address data structure specified for the
      socket */
   static struct sockaddr_in address  = { AF_INET };

   /* address of the remote client system */
   long remote_client_address;

   /* port number of the remote client system */
   short remote_client_port;

   /* size of the address data structure of the remote client
      system */
   int address_size;

   /* options word passed to socket() function */
```

```
int options;


/* begin the process of attempting to connect to the
   remote client system */

 /* initialize the options  */
options =  SO_ACCEPTCONN;

/* get a pointer to the remote client system's name */
ptr_client_name = &remote_client_name[0];

/* convert the remote client system name to its address */
remote_client_address = rhost(&ptr_client_name);

/* initialize the remote client port number above the
   system reserved ports */
remote_client_port = IPPORT_RESERVED + 1;

/* add the remote client port number to the number of
   reserved ports */
remote_client_port = remote_client_port + port_number;

/* initialize the remote client socket number to an
   invalid value */
remote_client_socket = -1;

/* remote client system address family (Internet in this
   case) */
address.sin_family = AF_INET ;

/* place the remote client port number into the address
   data structure */
address.sin_port = remote_client_port;

/* put the port number in network byte order */
address.sin_port = htons(address.sin_port);

/* place the remote client system's address in the address
   data structure */
address.sin_addr.s_addr = remote_client_address;

/* find number of bytes in the remote client address */
address_size = sizeof(remote_client_address);

/* attempt to open a local socket */
   local_server_socket =
      socket(SOCK_STREAM, (struct sockproto *)0, &address,
             options);
```

```
   if(local_server_socket < 0)
      {
      /* the server couldn't open a local socket */
      perror("Server couldn't open a local socket:");
      }

   else
      {

      printf("Server waiting to connect to
              %s\n",remote_client_name);

      /* attempt to accept a connection */
      remote_client_socket =
         accept(local_server_socket, &address);

      if(remote_client_socket < 0)
         {
         /* an error occurred in the server attempting to
            accept a connection from remote client system */
         perror("Server couldn't accept a connection from the
                 remote client system: ");

         close(local_server_socket);
         }
      else
         {
         /* the server accepted a connection from the remote
            client system */
         break;
         }

      } /* end else local_server_socket >= 0 */


   /* return the socket number of the remote client system */
   return(local_server_socket);

   } /* end of function connect_server() */



/*************************************************************

The connect_client(remote_server_name port_number) function
performs all the actions required to connect a client system
to a remote server system

*************************************************************/
```

```
int connect_client(remote_server_name, port_number)

   /* name and port number of the remote server system */
   char remote_server_name[];
   int  port_number;

   {

   /* pointer to the name of the remote server system */
   char *ptr_server_name;

   /* local socket number */
   int local_client_socket;

   /* function that opens a socket */
   int socket();

   /* function that connects local socket to remote server
      socket */
   int connect();

   /* function that returns the remote server system's
      address given a name - this is a System V function */
   long rhost();

   /* the socket number on the remote server system  */
   int remote_server_socket;

   /* the protocol and address data structure specified for
      the socket */
   static struct sockaddr_in address = { AF_INET };

   /* address of the remote server system */
   struct hostent *remote_server_address;

   /* port number of remote system */
   short remote_server_port;

   /* options word passed to socket() function */
   int options;


   /* begin the process of attempting to connect to the
      remote server system */

   /* maintain socket as long as possible */
   options =  SO_KEEPALIVE ;

   /* establish ptr to the remote server name */
   ptr_server_name = &remote_server_name[0];
```

```
/* convert the name of the remote server system to an
   address */
remote_server_address = rhost(&ptr_server_name);


/* initialize remote server port number above the system
   reserved ports */
remote_server_port = IPPORT_RESERVED + 1;

/* add the user's remote server port number to the number
   of reserved ports */
remote_server_port = remote_server_port + port_number;

/*  remote server system address family(Internet in this
    case) */
address.sin_family = AF_INET;

/* place the remote server port number into the address
   structure */
address.sin_port = remote_server_port;

/* put the port number in network byte order */
address.sin_port = htons(address.sin_port);

/* attempt to obtain a local socket */
local_client_socket =
   socket(SOCK_STREAM, (struct sockproto *)0, &address,
          options);

if(local_client_socket < 0)
   {
   /* the client couldn't open a local socket */
   perror("Client couldn't open a local socket: ");
   }
else
   {
   /* place Internet address family type in address
      structure */
   address.sin_family = AF_INET;

   /* attempt to connect local client socket to remote
      server socket */
   remote_server_socket =
      connect(local_client_socket, &address);

   if(remote_server_socket < 0)
      {
      /* an error occurred in attempting to connect to
         the remote server socket */
      perror("Client couldn't connect to the remote
              server socket: ");
```

```
            close(local_client_socket);
            }
         else
            {
            /* successfully connected to the remote server
               system */
            printf("Connection established with
                    %s.\n",remote_server_name);
            break;
            }

         } /* end else socket >= 0 */

      /* return the socket number of the local client system */
      return(local_client_socket);

      } /* end of funcition connect_client() */


/*************************************************************

function write_immediate(socket_number,buffer,nbytes) writes
to the network connection by forking a child process which
actually performs the write operation.

*************************************************************/


int write_immediate(socket_number, buffer_pointer, nbytes)

      /* socket number to be written */
      int socket_number;

      /* buffer of bytes to be written */
      char buffer[];

      /* the number of bytes to be written */
      long nbytes;

      {

      /* function which initiates a child process */
      int fork();

      /* value returned from routine fork()  */
      int forkval;


      /* initiate a child process which will perform the write
         operation */
      forkval = fork();
```

```
/* determine whether a child process was successfully
   started */
if(forkval == 0)
   {
   /* attempt to perform the write operation */
   while(write(local_socket,buffer,nbytes) != nbytes)
      {
      /* attempt to write the buffer contents */
      }
   /* terminate the child process after the buffer is
      successfully written */
   exit(1);
   }
else
   /* an error occurred in starting the child process */
   {
   perror("Error occurred while attempting to fork a
           write-immediate process: ");
   /* return an error value to the application */
   return(-1);
   }
/* return a value indicating successful operation */
return(forkval);
} /* end of write_immediate() function */
```

APPENDIX C

INTERTEXT PROGRAM LISTING

The source code for the Intertext program is contained in this appendix. It is written in the C language for UNIX systems. If a workstation with multiple programming shell windows is available, this program can be executed in a number of these windows simultaneously. This allows a user to participate in or monitor one or more conferences at one time.

```
/*

Program intertext.c
Version 04 November 1986

This program allows a user on a UNIX terminal to send
messages to another user on another UNIX terminal.

Link the object code from compiling this source file with
object file net4.2.o or netV.o.

*/

#include <stdio.h>
#define TRUE 1
#define FALSE 0

main()
   {
   /* socket number of the local system */
   int user_socket;

   /* function which connects a local server system to a
      remote client system */
   int connect_server();

   /* function which connects a local client system to a
      remote server system */
   int connect_client();

   /* client/server status input variable */
   char *client_server;
```

```
/* Boolean variable indicating client/server status */
int server;
/* name of remote system */
char *remote_system;

/* buffer for messages */
char *buf

/* number of bytes actually read/written and length of
   message buffer */
int nbytes, buflen;


/* program execution begins here */

/* determine remote system name and conference name */
puts("Please type the name of the remote system you
        would like to connect to: ");
gets(remote_system);

/* determine the port number */
puts("Please type the port number you would like to
        connect to: ");
scanf("%d",port_number);

/* determine whether user will be server or client */
puts("Are you initiating a conference or joining an
      existing one (please type I or J): ");
gets(client_server);
if((client_server[0] == 'i') || (client_server[0] == 'I'))
   server = TRUE;
else
   server = FALSE;

if(server == TRUE)
   /* open connection to the remote system as a server */
   user_socket = connect_server(remote_system,
                                port_number);
else
   /* open connection to the remote system as a client */
   user_socket = connect_client(remote_system,
                                port_number);

/* initialize the buffer size to a maximum of 256
   characters */
buflen = 256;

/* loop until the first character of a message is 'q' */
while(buf[0] != 'q')
   {
```

```
   /* if a server read a message and send it */
   if(server == TRUE)
      {
      /* read a line of characters from the user's
      keyboard into the buffer */
      gets(buf);

      /* write the contents of the buffer to the remote
         system */
      nbytes = write(user_socket,buf,buflen);

      /* read a message from the remote system into the
         buffer */
      nbytes = read(user_socket,buf,buflen);

      /* write the contents of the buffer to the screen */
      puts(buf);

      } /* end of if(server == TRUE) */
   else
      {
     /* read a message from the remote system into the
         buffer */
      nbytes = read(user_socket,buf,buflen);

      /* write the contents of the buffer to the screen */
      puts(buf);

      /* read a line of characters from the user's
      keyboard into the buffer */
      gets(buf);

      /* write the contents of the buffer to the remote
         system */
      nbytes = write(user_socket,buf,buflen);

      } /* end of else(server != TRUE) */

   } /* end of while(buf[0] != 'q') */

/* close the connection to the remote system */
close(user_socket);

} /* end of main() */
```

APPENDIX D

INTERDRAW PROGRAM LISTING

The source code for the Interdraw program is contained in this appendix. It is written in the C language for UNIX systems. If a workstation with multiple programming shell windows is available, this program can be executed in a number of these windows simultaneously. This allows a user to participate in or monitor one or more graphics conferences at one time.

```
/*

This is program Interdraw.c
Version 9 Dec 86

Interdraw.c is an IRIS-2400 program that allows two people on
two IRIS workstations to draw in a window which is displayed
on both of their graphics screens simultaneously.

Run this program under "mex," the Window Manager for the IRIS
by typing "mex", pressing the return key, and then typing
"interdraw."  Answer the questions about the remote system,
port number, and type "I" to initiate.  Then have another
user do the same procedure on another IRIS, answering "J" to
join.  A window will be drawn and filled with light blue
(cyan).  Move the cursor into the window and press the right
mouse button once.  Then move the cursor with the mouse to
move a "rubber band" line.  Draw a line by pressing the left
mouse button, move the origin of the line by pressing the
middle mouse button, and quit by pressing the right mouse
button.  Your lines will be drawn in red, while the other
user's lines will be drawn in blue.

Consult the IRIS 2400 User's Manual for details about the
graphics routines used in this program.

Link the object code from compiling this source file with
object file netV.o.

*/
```

```
/* include the IRIS graphics and device libraries */
#include "gl.h"
#include "device.h"

/* define the window world coordinate system */
#define WXMIN 0.0
#define WXMAX 1023.0
#define WYMIN 0.0
#define WYMAX 767.0

/* define command constants */
#define NOACTION 0
#define QUIT 999

   /* socket number of the local system */
   int user_socket;

   /* function which connects a local server system to a
      remote client system */
   int connect_server();

   /* function which connects a local client system to a
      remote server system */
   int connect_client();

   /* client/server status input variable */
   char *client_server;

   /* Boolean variable indicating client/server status */
   int server;

   /* name of remote system */
   char *remote_system;

   /* record of the last action taken */
   long local_action, remote_action;

   /* buffer for passed data */
   float buf[3];

   /* declare object names */
   Object local_single_line;
   Object local_accumulative;
   Object remote_single_line;
   Object remote_accumulative;

   /* declare tag names */
   Tag local_movetag,local_drawtag;
   Tag remote_movetag,remote_drawtag;

   /* x,y coords of local and remote lines */
```

```
    float local_x,local_y;
    float remote_x,remote_y;

    /* window origin, size, and cursor initial position */
    long origin_x, origin_y;
    long size_x, size_y;
    long init_x, init_y;

    /* writemask value */
    Colorindex wmask;

    /* mouse and button values */
    short value, value_x, value_y;

    /* mouse, button, and redraw device detection variables */
    Device active_device, temp_device;

    /* window identifier */
    int drawing_window;

    /* number of bytes actually written/read during a write or
       read operation */
    int nbytes;


main()
    {
    /* program execution begins here */

    /* determine remote system name and conference name */
    puts("Please type the name of the remote system you
            would like to connect to: ");
    gets(remote_system);

    /* determine the port number */
    puts("Please type the port number you would like to
            connect to: ");
    scanf("%d",port_number);

    /* determine whether user will be server or client */
    puts("Are you initiating a conference or joining an
          existing one (please type I or J): ");
    gets(client_server);
    if((client_server[0] == 'i') || (client_server[0] == 'I'))
       server = TRUE;
    else
       server = FALSE;

    if(server == TRUE)
       /* open connection to the remote system as a server */
       user_socket = connect_server(remote_system,
```

```
                                    port_number);
else
   /* open connection to the remote system as a client */
   user_socket = connect_client(remote_system,
                                    port_number);

/* initialize values for x,y */
local_x = WXMIN + (WXMAX - WXMIN)/2.0;
local_y = WYMIN + (WYMAX - WYMIN)/2.0;
remote_x = WXMIN + (WXMAX - WXMIN)/2.0;
remote_y = WYMIN + (WYMAX - WYMIN)/2.0;

/* initialize and open an IRIS Window Manager window */
prefposition(255,768,192,576);
foreground();
drawing_window = winopen("drawing_window");
wintitle("Drawing Window");

/* allow the user to move and resize the window */
winset(drawing_window);
winconstraints();

/* put the IRIS into double buffer mode */
doublebuffer();

/* configure the IRIS (means use the above command
   settings) */
gconfig();

/* enable all the bit planes for writing */
/* sets to 2**(getplanes()) minus one (all bits on) */
wmask = ((1<<getplanes()) -1);
writemask(wmask);

/* Full screen viewport */
viewport(0,1023,0,767);

/* orthogonal projection 2D for the world coord sys */
ortho2(0.0,1023.0,0.0,767.0);

/* determine initial size and location of the window */
getsize(&size_x, &size_y);
getorigin(&origin_x, &origin_y);

/* determine the center of the window */
init_x = origin_x + size_x/2;
init_y = origin_y + size_y/2;

/* set initial value and range limit for mouse */
setvaluator(MOUSEX,init_x,0,1023);
setvaluator(MOUSEY,init_y,0,767);
```

```
/* queue redraw events */
qdevice(REDRAW);

/* queue mouse button events */
qdevice(MOUSE1);
qdevice(MOUSE2);
qdevice(MOUSE3);
qdevice(MOUSEX);
qdevice(MOUSEY);

/* record mouse x and y coordinates on the device queue
   whenever the middle or right mouse buttons are
   pressed */
tie(MOUSE2,MOUSEX,MOUSEY);
tie(MOUSE3,MOUSEX,MOUSEY);

/* generate the local single line object */
local_single_line = genobj();

/* define the local single line object */
makeobj(local_single_line);

   /* make the lines 3 pixels thick */
   linewidth(3);

   /* generate and store a tag for moving the local single
   line starting point */
   local_movetag = gentag();
   maketag(local_movetag);

   /* move the local single line starting point */
   move2(local_x,local_y);

   /* generate and store a tag for moving the local single
   line ending point */
   local_drawtag = gentag();
   maketag(local_drawtag);

   /* move the local single line ending point */
   draw2(local_x,local_y);

closeobj();

/* generate the remote single line object */
remote_single_line = genobj();

/* define the remote single line object */
makeobj(remote_single_line);

   /* make the lines 3 pixels thick */
```

```
    linewidth(3);

    /* generate and store a tag for moving the remote
    single line starting point */
    remote_movetag = gentag();
    maketag(remote_movetag);

    /* move the remote single line starting point */
    move2(remote_x,remote_y);

    /* generate and store a tag for moving the local single
    line ending point */
    remote_drawtag = gentag();
    maketag(remote_drawtag);

    /* move the remote single line ending point */
    draw2(remote_x,remote_y);

closeobj();

/* generate the object which holds the accumulated
   local drawing */
local_accumulative = genobj();

/* define the local accumulative object */
makeobj(local_accumulative);

    /* make the lines 3 pixels thick */
    linewidth(3);

  /* add a move to the local accumulative object */
  move2(local_x,local_y);

closeobj();

/* generate the object which holds the accumulated
   remote drawing */
remote_accumulative = genobj();

/* define the remote accumulative object */
makeobj(remote_accumulative);

    /* make the lines 3 pixels thick */
    linewidth(3);

    /* add a move to the remote accumulative object */
    move2(remote_x,remote_y);

closeobj();

/* turn on the default cursor */
```

```
setcursor(0,RED,wmask);

/* attach the cursor to the mouse */
attachcursor(MOUSEX,MOUSEY);

/* turn the cursor on */
curson();

/* if a server system, start communications by sending a
   record to the remote system */
if(server == TRUE)
   {
   buf[0] = local_action;
   buf[1] = local_x;
   buf[2] = local_y;
   nbytes = write(server_socket,buf,3*sizeof(float));
   }

/* main loop */
while(TRUE)
   {
   /* process queue data */
   while(qtest() != 0)
      {
      /* read a value from the queue */
      active_device = qread(&value);

      /* determine whether a window has been moved */
      if(active_device == REDRAW)
         {
         reshapeviewport();
         getorigin(&origin_x,&origin_y);
         getsize(&size_x, &size_y);

         init_x = ((size_x - 1)/(WXMAX - WXMIN)) *
                  (local_x - WXMIN) + origin_x;
         init_y = ((size_y - 1)/(WYMAX - WYMIN)) *
                  (local_y - WYMIN) + origin_y;

         setvaluator(MOUSEX, init_x, 0, 1023);
         setvaluator(MOUSEY, init_y, 0, 767);
         }


      /* start out by saying no action has occurred */
      local_action = NOACTION;

      /* should we start a new line? */
      if(active_device == MOUSE2)
         {
         /* middle mouse button pressed */
```

```
        local_action = MOUSE2;

        /* start a new line at the cursor position */
        /* dont add the dragged line to local
           accumulative object */
        /* read mouse x and y values from the queue */
        temp_device = qread(&value_x);
        temp_device = qread(&value_y);

        local_x = ((WXMAX - WXMIN)/(size_x - 1)) *
                  (value_x - origin_x) + WXMIN;
        local_y = ((WYMAX - WYMIN)/(size_y - 1)) *
                  (value_y - origin_y) + WYMIN;

        /* update the local single line object */
        editobj(local_single_line);

           objreplace(local_movetag);
           move2(local_x,local_y);

           objreplace(local_drawtag);
           draw2(local_x,local_y);

        closeobj();

/* add move2 command to the local accumulative object */
editobj(local_accumulative);

   move2(local_x,local_y);

closeobj();

        }

     /* should we add a line to the local accumulative
        object? */
     if(active_device == MOUSE3)
        {

        /* left mouse button pressed */
        local_action = MOUSE3;

        /* add line to the local_accumulative object */
        temp_device = qread(&value_x);
        temp_device = qread(&value_y);

        local_x = ((WXMAX - WXMIN)/(size_x - 1)) *
                  (value_x - origin_x) + WXMIN;
        local_y = ((WYMAX - WYMIN)/(size_y - 1)) *
                  (value_y - origin_y) + WYMIN;
```

```
      /* add the current line to the local accumulative
         object */
      editobj(local_accumulative);

         draw2(local_x,local_y);

      closeobj();

      /* write x,y over local_movetag of single line
         object */
      editobj(local_single_line);

         objreplace(local_movetag);
         move2(local_x,local_y);

      closeobj();

      }

   /* if mouse is moved in the x direction */
   if(active_device == MOUSEX)
      {
      local_x = ((WXMAX - WXMIN)/(size_x - 1)) *
                (value - origin_x) + WXMIN;

      /* update the endpoint of the current line */
      editobj(local_single_line);

         objreplace(local_drawtag);
         draw2(local_x,local_y);

      closeobj();
      }

   /* if mouse is moved in the y direction */
   if(active_device == MOUSEY)
      {
      local_y = ((WYMAX - WYMIN)/(size_y - 1)) *
                (value - origin_y) + WYMIN;

      /* update the endpoint of the current line */
      editobj(local_single_line);

         objreplace(local_drawtag);
         draw2(local_x,local_y);

      closeobj();
      }


   } /* end if(qtest() != 0) */
```

```
/* redraw the entire window's contents */

/* clear the window to cyan */
color(CYAN);
clear();

/* receive a record from the remote system */
nbytes = read(server_socket,buf,3*sizeof(float));
remote_action = buf[0];
remote_x = buf[1];
remote_y = buf[2];

switch(remote_action)
   {

   case QUIT:

      /* quit because right mouse button was pressed */
      break;

   case MOUSE2:

      /* start a new line at the cursor position */
      /* dont add the dragged line to accumulative */
      editobj(remote_single_line);

         objreplace(remote_movetag);
         move2(remote_x,remote_y);

         objreplace(remote_drawtag);
         draw2(remote_x,remote_y);

      closeobj();

      /* add move2 command to accumulative object */
      editobj(remote_accumulative);

         move2(remote_x,remote_y);

      closeobj();

      break;

   case MOUSE3:

      /* add line to the accumulative object */

      editobj(remote_accumulative);

         draw2(remote_x,remote_y);
```

```
            closeobj();

            /* write x,y over movetag of single line
               object */
            editobj(remote_single_line);

               objreplace(remote_movetag);
               move2(remote_x,remote_y);

            closeobj();

            break;

         default:

            break;

         } /* end switch(remote_action) */

      /* move the draw part of the remote single line */
      editobj(remote_single_line);

         objreplace(remote_drawtag);
         draw2(remote_x,remote_y);

      closeobj();

      /* draw the remote single line */
      color(BLUE);
      callobj(remote_single_line);

      /* draw the remote accumulative lines buffer */
      callobj(remote_accumulative);


      /* move the draw part of local single line */
      editobj(local_single_line);

         objreplace(local_drawtag);
         draw2(local_x,local_y);

      closeobj();

      /* draw the local single line */
      color(RED);
      callobj(local_single_line);

      /* draw the local accumulative lines buffer */
      callobj(local_accumulative);
```

```
    /* send a record to the remote system */
    buf[0] = local_action;
    buf[1] = local_x;
    buf[2] = local_y;
    nbytes = write(server_socket,buf,3*sizeof(float));

    /* change the graphics buffers for smooth animation */
    swapbuffers();

    /* quit if right mouse button is pressed */
    if(active_device == MOUSE1)
       {
       buf[0] = QUIT;
       write(server_socket,buf,3*sizeof(float));
       break;
       }

    /* if remote system quits then quit also */
    if(remote_action == QUIT)
       {
       break;
       }

    } /* end of while(TRUE) */

/* perform some clean-up by setting graphics to black */
color(BLACK);
clear();
swapbuffers();
clear();
curson();
textinit();
swapbuffers();
finish();

/* graphics exit */
gexit();

/* close the network connection */
close(user_socket);

} /* end of main() */
```

LIST OF REFERENCES

1. Sarin, S. and Greif, I., "Computer-Based Real-Time Conferencing Systems," IEEE Computer, p. 34, October 1985.

2. Poggio, A., and others, "CCWS: A Computer-Based, Multimedia Information System," IEEE Computer, pp. 92-103, October 1985.

3. Aguilar, L., "A Format for a Graphical Communications Protocol," IEEE Computer Graphics and Applications, pp. 52-62, March 1986.

4. Shuey, D., Bailey, D., and Morrissey, T., "PHIGS: A Standard, Dynamic, Interactive Graphics Interface," IEEE Computer Graphics and Applications, pp. 50-57, August 1986.

5. Herman, W., "The Cost of Graphics Standards," Computer Graphics World, p. 128, November 1986.

BIBLIOGRAPHY

Apple Computer, Inc., Inside Macintosh, 1985.

Foley, J. and Van Dam, A., Fundamentals of Interactive Computer Graphics, Addison-Wesley, 1982.

International Organization for Standards, GKS-3D Functional Description, ISO Second DP 8805, October 1985.

International Organization for Standards, GKS Functional Description, ISO IS 7942, July 1985.

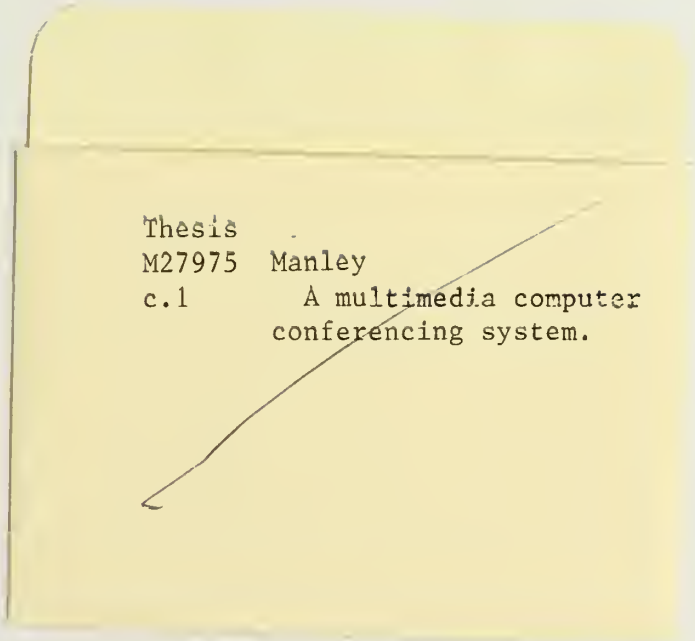
Leffler, S., Fabry, R., and Joy, W., A 4.2 BSD Interprocess Communication Primer, draft documentation for UNIX 4.2 Berkeley Software Distribution, 11 August 1986.

INITIAL DISTRIBUTION LIST

	No. Copies
1. Defense Technical Information Center Cameron Station Alexandria, Virginia 22304-6145	2
2. Library, Code 0142 Naval Postgraduate School Monterey, California 93943-5002	2
3. Chairman, Code 52 Department of Computer Science Naval Postgraduate School Monterey, California 93943-5000	1
4. Computer Technology Curricular Officer, Code 37 Naval Postgraduate School Monterey, California 93943-5000	1
5. Dr. Michael J. Zyda, Code 52Zk Department of Computer Science Naval Postgraduate School Monterey, California 93943-5000	1
6. LCDR James E. Manley 12465 Hedges Run Drive Woodbridge, VA 22192	1

17898 2

DUDLEY KNOX LIBRARY
NAVAL POSTGRADUATE SCHOOL
MONTEREY, CALIFORNIA 93943-5



thesM27975
A multimedia computer conferencing syste
3 2768 000 70772 3
DUDLEY KNOX LIBRARY